UNIVERSIDADE da MADEIRA

CTeSP CURSOS TÉCNICOS SUPERIORES PROFISSIONAIS

Tecnologias e Programação de Sistemas de Informação

Enumerados e Coleções

Arquitetura de Dispositivos | David Jardim

Cofinanciado por:

MADEIRA 14-20

REGIÃO AUTÓNOMA DA MADEIRA

PORTUGAL 2020

UNIÃO EUROPEIA
Fundo Social Europeu

Da aula anterior…

- Herança e Polimorfismo

Enumerado

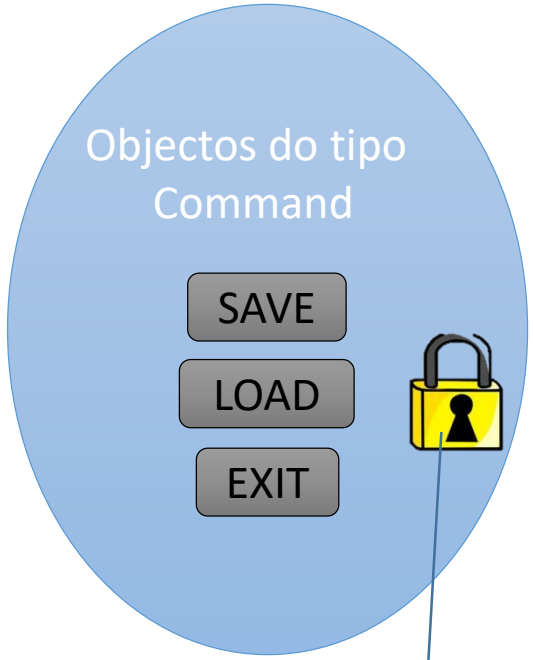
- Tipo com conjunto fixo e finito de valores
  - Exemplos: dias da semana, direcções, estado civil
- Podem ter atributos e construtores
- Têm operações associadas
- Melhores que inteiros ou cadeias de caracteres para representar pequenos conjuntos

Exemplo (opções de menu)

```
...
Scanner scanner = new Scanner(System.in);
System.out.println("Introduza um comando:");
String command = scanner.nextLine();
if(command.equals("SAVE")) {
  // gravar...
}
else if(command.equals("LOAD")) {
  // carregar...
}
else if(command.equals("EXIT")) {
  // sair...
}
...
```

Exemplo (opções de menu)

```
public enum Command {
   SAVE, LOAD, EXIT;
}

...
Scanner scanner = new Scanner(System.in);
System.out.println("Introduza uma comando:");
String line = scanner.nextLine();
Command command = Command.valueOf(line);
if(command == Command.SAVE) {
  // gravar...
}
else if(command == Command.LOAD) {
  // carregar...
}
else if(command == Command.EXIT) {
  // sair...
}
...
```



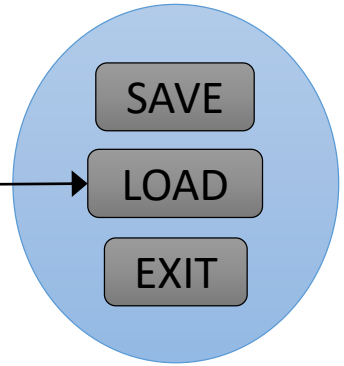
"Conjunto fixo"... não é possível remover ou adicionar objectos em tempo de execução

Operação valueOf(String)

- Disponível em todos os tipos enumerados
- Obtem o elemento de um enumerado dado o seu nome (objecto String)

```
public enum Command {
    SAVE, LOAD, EXIT;
}
```

```
Command c = Command.valueOf("LOAD");
```



Instrução de seleção switch

- Alternativa ao if-else
  - Adequada quando as diferentes alternativas de execução são determinadas pelo valor de determinada variável
    - A variável pode ter um dos tipos primitivos numéricos para representar inteiros (byte, short, int) , char, ou ser de um tipo enumerado

Exemplo (opções de menu / switch)

```
public enum Command {
    SAVE, LOAD, EXIT;
}

...
Scanner scanner = new Scanner(System.in);
System.out.println("Introduza um comando:");
String line = scanner.nextLine();
Command command = Command.valueOf(line);
switch(command) {
  case SAVE:
     // gravar...
     break;
  case LOAD:
     // carregar...
     break;
  case EXIT:
     // sair...
     break;
}
```

Exemplo (direção)

```
public class Direction {
    private String name;

    public Direction(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Fará sentido existirem outras instâncias para além destas?

```
Direction north = new Direction("North");
Direction south = new Direction("South");
Direction east = new Direction("East");
Direction west = new Direction("West");
```

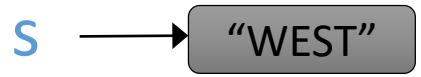
Exemplo (direção)

```
public enum Direction {
   NORTH, SOUTH, EAST, WEST;

   public String prettyName() {
      return name().charAt(0) +
             name().substring(1).toLowerCase();
   }
}

String s1 = Direction.NORTH.name();
System.out.println(s1);
String s2 = Direction.SOUTH.prettyName();
System.out.println(s2);


> NORTH
> South
```

Operação name()

- Disponível em todos os tipos enumerados
- Devolve um objecto String com o identificador do elemento do enumerado

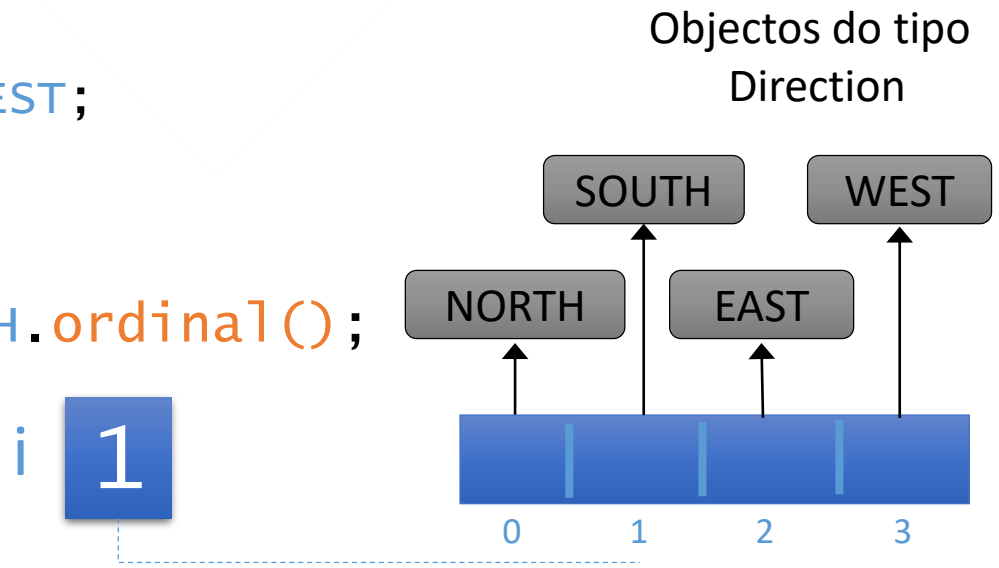
```
String s = Direction.WEST.name();
```



Operação ordinal()

- Disponível em todos os tipos enumerados
- Devolve o índice do elemento do enumerado de acordo com a ordem de declaração

```
public enum Direction {
   NORTH, SOUTH, EAST, WEST;
   ...
}

int i = Direction.SOUTH.ordinal();
```

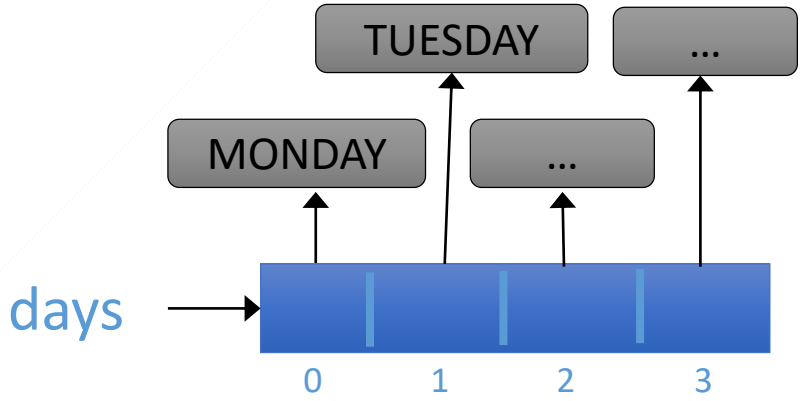


Operação values()

- Disponível em todos os tipos enumerados
- Devolve um vector com todos os elementos do enumerado (pela ordem que são declarados)

```
WeekDay[] days = WeekDay.values();

for(int i = 0; i < days.length; i++) {
   WeekDay day = days[i];
   String name = day.name();
   System.out.println(name);
}
```

```
> MONDAY
> TUESDAY
> WEDNESDAY
> THURSDAY
> FRIDAY
> SATURDAY
> SUNDAY
```



Infra-estrutura aplicacional de colecções do Java

- Infra-estrutura aplicacional englobando interfaces, classes abstractas e concretas, e algoritmos que disponibilizam vários tipos de colecção em Java

Java Collections Framework (JCF)

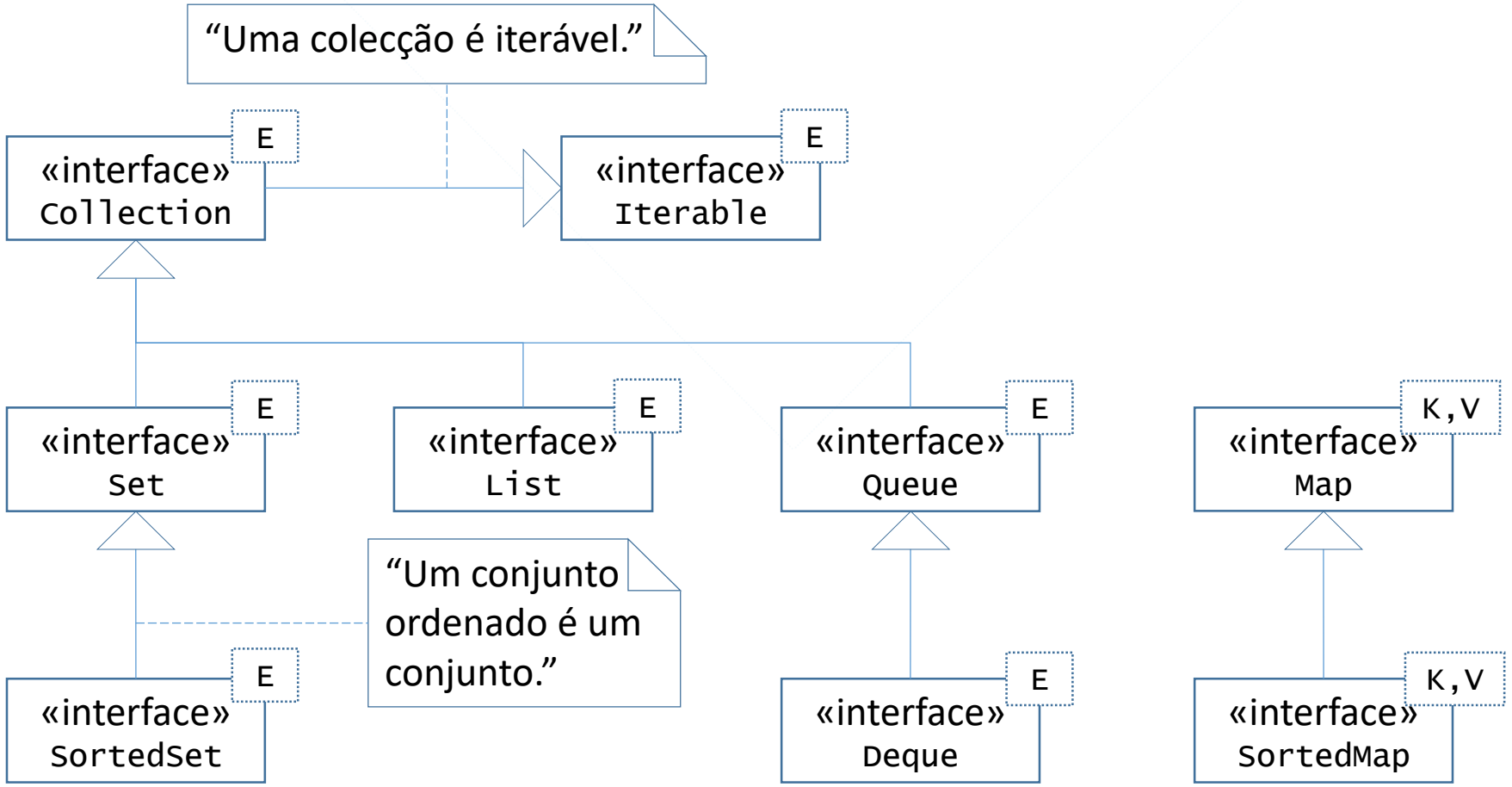
- Colecções
  - Agregados estruturados de elementos
  - Cada tipo de colecção tem propriedades específicas
  - Têm diferentes eficiências a realizar operações equivalentes

JCF: tipos de colecção

Tipo	Natureza	Repetições	Ordenado	Tipo de ordem
`Set<E>`	Conjunto	não	?	?
`List<E>`	Sequência	sim	sim	de inserção
`Queue<E>`	Fila de espera	sim	sim	extração: sim, internamente: ?
`Stack<E>`	Pilha	sim	sim	extração: sim, internamente: ?
`Map<K,V>`	Mapeia chaves em valores	não (chaves) sim (valores)	?	?

Legenda:
`E` – tipo dos elementos
`K` – tipo das chaves de um mapa
`V` – tipo dos valores de um mapa
? – característica depende do tipo concreto

JCF: principais interfaces



JFC: estruturas de dados subjacentes

Nome	Nome (inglês)	Descrição
Vector	*Array*	Sequência de elementos contíguos em memória, com indexação muito rápida mas inserção de novos elementos lenta (excepto nos extremos quando não é necessário um aumento da capacidade).
Lista ligada	*Linked list*	Sequência de elementos ligados, com indexação e pesquisa lentos mas inserção rápida em qualquer local.
Árvore	*Tree*	Sequência de elementos organizados em árvore, com todas as operações essenciais razoavelmente rápidas.
Tabela de dispersão	*Hash(ing) table*	Elementos espalhados em matriz usando índices obtidos aplicando-lhes uma função de endereçamento, com todas as operações essenciais muito rápidas (troca mais velocidade por maior consumo de memória).

JCF: elementos, chaves e valores

- Têm de implementar
  - `boolean equals(Object another)`
  - `int hashCode()`

Para procurar.

Para tabelas de dispersão.

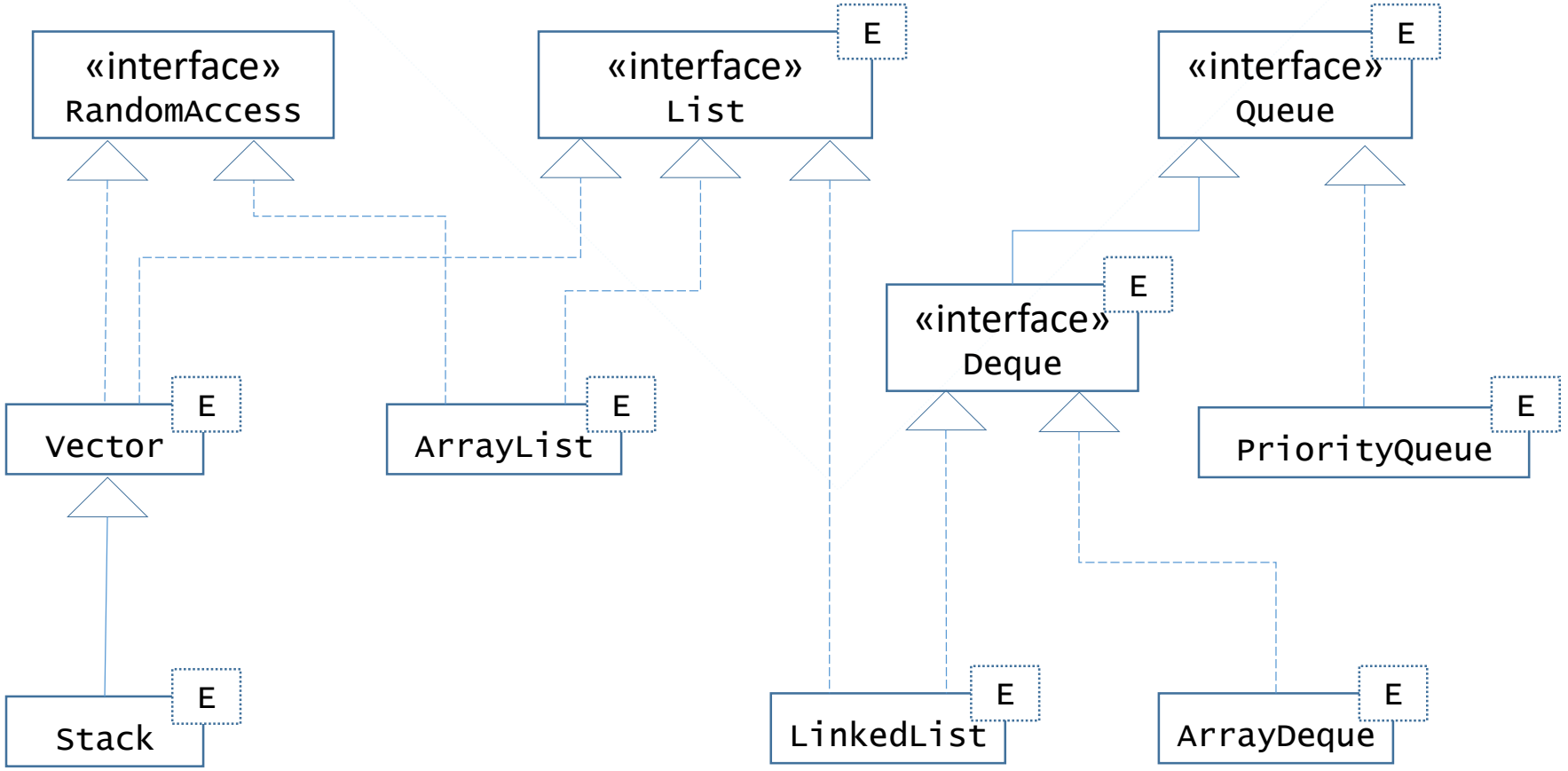
- Operações são fornecidas pela classe `Object`!

- Podem ser sobrepostas (*com cuidado*)
  - Se
    `one.equals(another)`
    então
    `one.hashCode() == another.hashCode()`
  - Outras restrições

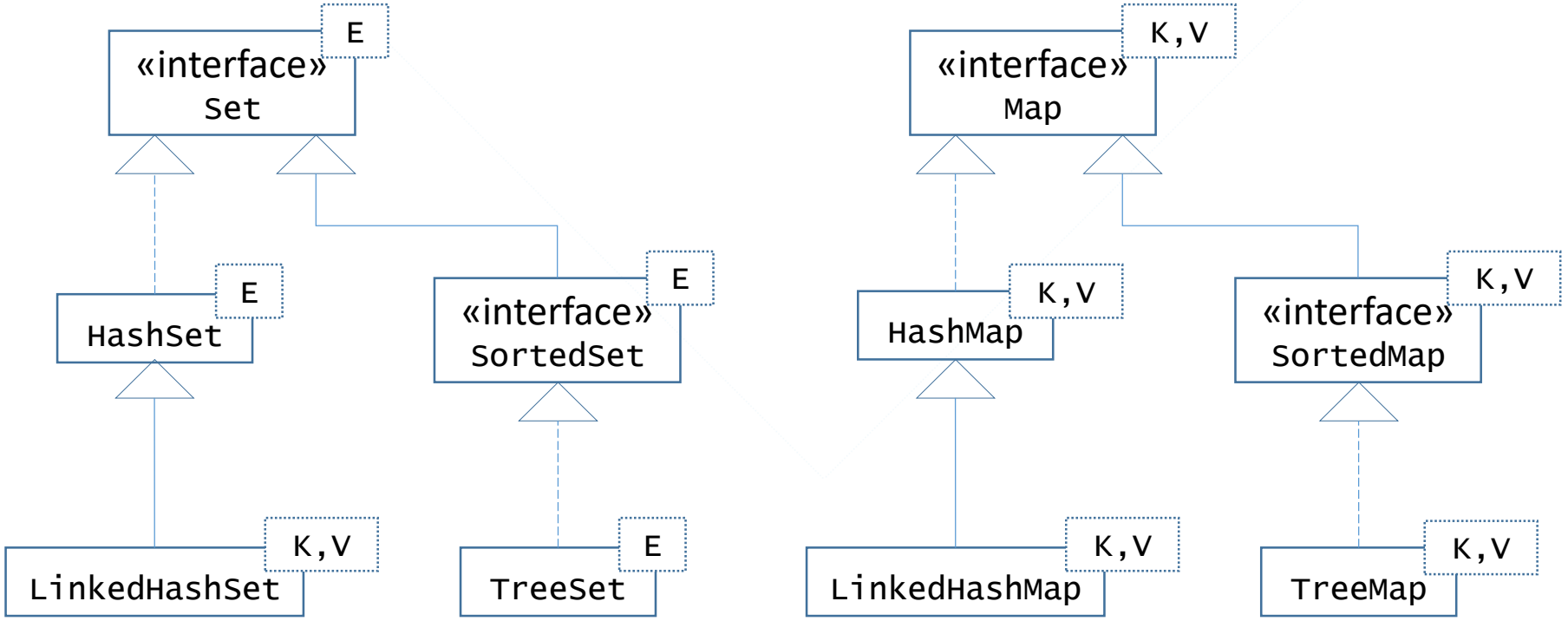
JCF: classes concretas

Tipo	Representação interna	Restrições adicionais
`ArrayList<E>`	Vector	-
`Vector<E>`	Vector	-
`LinkedList<E>`	Lista ligada	-
`ArrayDeque<E>`	Vector	-
`Stack<E>`	Vector (via `Vector<E>`)	-
`PriorityQueue<E>`	Vector (organizada como árvore)	`E` implementa `Comparable<E>`
`TreeSet<E>`	Árvore	`E` implementa `Comparable<E>`
`TreeMap<K,V>`	Árvore	`K` implementa `Comparable<K>`
`HashSet<E>`	Tabela de dispersão	-
`HashMap<K,V>`	Tabela de dispersão	-

JCF: classes concretas



JCF: classes concretas



JCF: one.compareTo(another)

Relação entre one e another	Resultado da operação
one < another	< 0
one = another	= 0
one > another	> 0

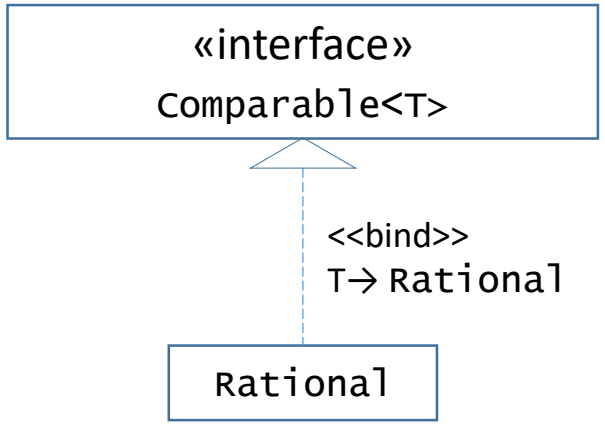
JCF: Boas práticas

- Classe implementa `compareTo`?
- Logo, *deve sobrepor* a sua especialização de `equals`…
- …pois por omissão `equals` compara *identidade* e não *igualdade*!
- As operações `compareTo` e `equals` devem ser consistentes…
- …ou seja, `one.compareTo(another) == 0` deve resultar no mesmo que `one.equals(another)`

Aplicação à classe Rational

```
public class Rational implements Comparable<Rational> {
    private final int numerator;
    private final int denominator;
    …

    public int compareTo(final Rational another){
        return getNumerator() * another.getDenominator()
             - another.getNumerator() * getDenominator();
    }

    …
}
```

Esta implementação só está correta se convencionarmos que o denominador é sempre positivo. Neste caso, isso deveria fazer parte da condição invariante.



Aplicação à classe Rational

```
public class Rational implements Comparable<Rational> {
    …

    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;

        final Rational other = (Rational) obj;

        return denominator == other.denominator && numerator == other.numerator;
    }
}
```

Aplicação à classe Rational

```
public class Rational implements Comparable<Rational> {
    private final int numerator;
    private final int denominator;
    …

    public int hashCode() {
        return Objects.hash(numerator, denominator);
    }

    …
}
```

Aplicação à classe Aluno

Aluno não tem de estar ordenado sempre da mesma forma (não tem uma ordem natural)

Para ter uma ordem alfabética por nome podemos definir:

```
public class ComparadorDeAlunos implements Comparator<Aluno> {

    public int compare(Aluno aluno1, Aluno aluno2) {
        return aluno1.getNome().compareTo(aluno2.getNome());
   }

}
```

Classe pacote Collections

```
List<Rational> racionais = new ArrayList<Rational>();
…
Collections.sort(racionais);
```

sort(List), Ordenar segundo a ordem natural (**Comparable**)

```
List<Aluno> alunos = new LinkedList<Aluno>();
…
Collections.sort(alunos, new  ComparadorDeAlunos());
```

sort(List, Comparator) , Ordenar segundo um critério

JCF: List e ArrayList

```
List<Course> courses = new ArrayList<Course>();
Course arqd = new Course("Arqd");
Course redes = new Course("Redes");

courses.add(arqd);   // adiciona ao fim
courses.add(redes);

int indexOfCourseToRemove = -1;

for (int i = 0; i != courses.size(); i++)
    if (courses.get(i) == redes)
        indexOfCourseToRemove = i;

if (indexOfCourseToRemove != -1)
    courses.remove(indexOfCourseToRemove);
courses.remove(arqd);
```

É comum usar um tipo mais genérico para aceder a uma colecção do que a classe real do objecto referenciado. Dessa forma pode-se alterar essa classe alterando apenas uma linha de código.

Remoção fora do ciclo? O.K.
Remoção dentro do ciclo? Bronca!

JCF: Vector

```
Vector<Course> courses = new Vector<Course>();

Course arqd = new Course("arqd");
Course redes = new Course("Redes");

courses.add(arqd);   // adiciona ao fim
courses.add(redes);

for (int i = 0; i != courses.size(); i++)
    out.println(courses.get(i));
```

JCF: Stack

```
Stack<Course> courses = new Stack<Course>();

Course arqd = new Course(“Arqd");
Course redes = new Course("Redes");

courses.push(arqd);   // adiciona no topo
courses.push(redes);

while (!courses.isEmpty()) {
    out.println(courses.peek());
    courses.pop();
}
```

JCF: List, LinkedList e Iterator

```
List<Course> courses = new LinkedList<Course>();
Course wb = new Course("WEB-Backend");

Iterator<Course> iterator =
    courses.iterator();

while (iterator.hasNext()) {
    Course course = iterator.next();
    if (course == wb)
        iterator.remove();
}
```

Quando possível deve usar-se o interface e não o tipo específico.

Dois em um: avança e devolve.

Remoção segura: É removido o último elemento devolvido por `next()`.

JCF: Queue e LinkedList

```
Queue<String> courseNames = new LinkedList<String>();

courseNames.add("Redes");
courseNames.add("WEB-Backend");
courseNames.add("Arqd");

while(!courseNames.isEmpty()) {
    out.println(courseNames.element());
    courseNames.remove();
}
```

JCF: Queue e LinkedList

```
Queue<Course> courses = new LinkedList<Course>();
Course arqd = new Course("Arqd");
Course redes = new Course("Redes");

courses.add(arqd); // adiciona ao início
courses.add(redes); // adiciona ao início

out.println(courses.element());
out.println(courses.element());

Iterator<Course> iterator = courses.iterator();
while (iterator.hasNext()) {
    Course course = iterator.next();
    out.println(course);
}
```

Mais uma vez, dois em um…

JCF: LinkedList e Deque

```
Deque<Course> courses = new LinkedList<Course>();
Course arqd = new Course("arqd");
Course redes = new Course("Redes");

courses.addFirst(arqd); // adiciona ao início
courses.addLast(redes); // adiciona ao fim

out.println(courses.getFirst());
out.println(courses.getLast());

Iterator<Course> iterator = courses.iterator();
while (iterator.hasNext()) {
    Course course = iterator.next();
    out.println(course);
}
```

A Deque permite adicionar e remover em ambas as extremidades

Ciclo for-each

```
List<Course> courses = new LinkedList<Course>();

for (Course course : courses)
    out.println(course);
```

Modo de iteração compacto, sem usar iterador, mas … de utilização limitada (não se pode alterar a colecção, não se pode facilmente percorrer subsequências da colecção, etc.).

JCF: Iteração e alteração concorrentes

```
List<Course> courses = new LinkedList<Course>();
Course redes = new Course("Redes");
…

for (Course course : courses) {
    courses.remove(redes);
    out.println(course);
}
```

Alterações durante o ciclo produzem resultados inesperados. Pode mesmo ser lançada a excepção `ConcurrentModificationException`.

JCF: Map e HashMap

```
Map<String, Course> courses = new HashMap<String, Course>();
courses.put("arqd", new Course("Arquitetura de …"));

if (courses.containsKey("arqd"))
    out.println(courses.get("arqd"));

for (String key : courses.keySet())
    out.println(key);

for (Map.Entry<String, Course> entry : courses.entrySet())
    out.println(entry);

for (Course course : courses.values())
    out.println(course);
```

JCF: Map e TreeMap

```
Map<String, Course> courses = new TreeMap<String, Course>();
courses.put("arqd", new Course("Arquitetura de …"));

if (courses.containsKey("arqd"))
    out.println(courses.get("arqd"));

for (String key : courses.keySet())
    out.println(key);

for (Map.Entry<String, Course> entry : courses.entrySet())
    out.println(entry);

for (Course course : courses.values())
    out.println(course);
```

JCF: Boas práticas na utilização de colecções

- Não usar coleções de `Object`
- Usar o tipo de coleção mais adequado
- Atentar na diferente eficiência das mesmas operações em diferentes tipos de coleção (consultar a documentação)

JCF: Boas práticas na utilização de colecções

- Não alterar uma coleção durante uma iteração ao longo dos elementos (ou usar o iterador para o fazer)
- Alteração de elementos de colecções com ordem intrínseca pode ter efeitos inesperados
- Ter atenção à documentação: nem todas as colecções permitem a inserção de elementos nulos


UNIVERSIDADE da MADEIRA
CTeSP
CURSOS TÉCNICOS
SUPERIORES PROFISSIONAIS